Chapter 3: Matrices, Calculations and Logical operations

3.1 Matrices
As mentioned in the earlier chapter, you can have lists of numbers as well as of letters. These list of numbers can be one-dimensional, in which case they called a *vector*. When they are two, three, or more-dimensional they are called a *matrix*.

\[
\text{mat1} = \begin{bmatrix} 1 & 54 & 3 \\ 2 & 1 & 5 \\ 7 & 9 & 0 \\ 0 & 1 & 0 \end{bmatrix}
\]

\[
\text{mat2} = \begin{bmatrix} 1 & 54 & 3 \\ 2 & 1 & 5 \\ 7 & 9 & 0 \\ 0 & 1 & 0 \end{bmatrix}
\]

Take a look at \text{mat1} and \text{mat2}. As you can see there is more than one way of entering a matrix. \text{mat1} and \text{mat2} were entered differently, but both have 4 rows and 3 columns.

When entering matrices a semi-colon is the equivalent of a new line.

You can find the size of matrices using the command \texttt{size}.

\[
\text{size(mat1)}
\]

\[
\text{ans} = \begin{bmatrix} 4 & 3 \end{bmatrix}
\]

For a two dimensional matrix the first value in \texttt{size} is the number of rows. The second value of size is the number of columns.

Now try:

\[
\text{vect1} = \begin{bmatrix} 1 & 2 & 4 & 6 & 3 \end{bmatrix}
\]

\[
\text{vect2} = \text{vect1}'
\]

\[
\text{vect1} = \begin{bmatrix} 1 & 2 & 4 & 6 & 3 \end{bmatrix}
\]

\[
\text{vect2} = \begin{bmatrix} 1 \\ 2 \\ 4 \end{bmatrix}
\]
Vectors can be tall instead of long, ‘ (the little symbol below the double quote on your keyboard) is a transpose, and allows you to swop rows and columns of a vector or a matrix.

Remember the command whos which will tell you the size of all your variables at once? Use whos to look at the size of mat1, mat2, vect1 and vect2.

**whos**

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>2x3</td>
<td>48</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>SPM5Path</td>
<td>1x34</td>
<td>68</td>
<td>char</td>
<td></td>
</tr>
<tr>
<td>V1</td>
<td>1x4</td>
<td>32</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>V2</td>
<td>1x4</td>
<td>32</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>ans</td>
<td>1x2</td>
<td>16</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>mat</td>
<td>3x5</td>
<td>120</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>mat1</td>
<td>4x3</td>
<td>96</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>mat2</td>
<td>4x3</td>
<td>96</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>vect1</td>
<td>1x5</td>
<td>40</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>vect2</td>
<td>5x1</td>
<td>40</td>
<td>double</td>
<td></td>
</tr>
</tbody>
</table>

### 3.2 Addition and subtraction

You can perform various calculations on matrices and arrays. You can add a single number (generally called a *scalar*) to a vector.

**vect1+3**

```
ans =
     4     5     7     9     6
```

You can subtract a scalar.

**vect2-3**

```
ans =
    -2
   -1
    1
    3
    0
```

### Scalar?

In physics a scalar is a value that only has magnitude (and not direction). In programming a scalar is defined as a quantity that can only hold a single value at a time, i.e. a single number or character. Generally the expression *scalar* tends to be used to refer to numbers rather than characters.
Matrix dimensions must agree?

You get this error if you try to add vectors of inappropriate sizes. To successfully add two vectors they must be the same orientation as well as the same length – i.e. you can’t add a tall thin vector to a short fat vector. The same goes for other operations (subtraction, point-wise multiplication & division and matrix multiplication & division) and applies to matrices as well as vectors.

When you get this error the first thing you should do is check the size of all the variables that you are trying to manipulate, and try to work out why Matlab thinks they are mismatched in size or shape. Often it’s the case that simply transposing one of the variables is all you need to do. This is an important thing to understand, so play with a few examples of your own and make sure you understand this.

You can add a vector onto itself

\[
\text{vect1} + \text{vect1}
\]

\[
\text{ans} = \\
2 \quad 4 \quad 8 \quad 12 \quad 6
\]

You can also add two vectors as long as they are the same size. You can’t add \text{vect1} and \text{vect2} together since they are different sizes.

\[
\text{vect1} + \text{vect2}
\]

??? Error using ==> plus
Matrix dimensions must agree.

\[
\text{vect3} = [1 \ 2 \ 3 \ 4] \\
\text{vect4} = [1 \ 3 \ 5 \ 2]';
\]

\[
\text{vect3} + \text{vect4};
\]

??? Error using ==> plus
Matrix dimensions must agree.

\[
\text{mat1} = [1 \ 2 \ 3; 4 \ 5 \ 6]; \\
\text{mat2} = [1 \ 2; 3 \ 4; 5 \ 6];
\]

\[
\text{mat1} - \text{mat2}'
\]

\[
\text{ans} = \\
0 \quad -1 \quad -2 \\
2 \quad 1 \quad 0
\]

\[
\text{mat1} - \text{mat2}
\]
3.3 Scalar multiplication and division
Here you simply use the symbols * and /. You can multiply a scalar with another scalar, with a vector, or with a matrix.

\[ 2 \times 3 \]
\[ \text{ans} = 6 \]

\[ \frac{2}{3} \]
\[ \text{ans} = 0.6667 \]

\[ \text{vect1} \times 3 \]
\[ \text{ans} = \begin{bmatrix} 3 & 6 & 12 & 18 & 9 \end{bmatrix} \]

\[ \text{vect1} \div 2 \]
\[ \text{ans} = \begin{bmatrix} 0.5 & 1 & 2 & 3 & 1.5 \end{bmatrix} \]

\[ \text{mat1} \times 0.5 \]
\[ \text{ans} = \begin{bmatrix} 0.5 & 1 & 1.5 \\ 2 & 2.5 & 3 \end{bmatrix} \]

\[ 3 + 1 \times 4 \]
\[ \text{ans} = 7 \]

\[ 3 \times 4 + 1 \]
\[ \text{ans} = 13 \]

Multiplication & division has priority over addition & subtraction. So the statements above are the equivalent of \(3+(1\times4)\) and \((3\times4)+1\). If you want to do addition or subtraction before multiplication or division you need to use brackets.

\[(3+1) \times 4 \]
\[ \text{ans} = 16 \]
3*(4+1)

ans =
15

Vector multiplication and division

As you may or may not remember from high school math there are actually two sorts of multiplication and two sorts of division when you are multiplying vectors with each other (or matrices with each other). They are called point-wise and inner-product multiplication respectively.

3.4 Point-wise vector multiplication and division

Point-wise (or element by element) multiplication and division is the simplest. In Matlab these are computed using .* and ./ (notice the period character).

V1=[1 2 3 4];
V2=[2 3 4 5];
V1.*V2
ans =
2 6 12 20

V2.*V1
ans =
2 6 12 20

V1./V2
ans =
0.5000 0.6667 0.7500 0.8000

V2./V1
ans =
2.0000 1.5000 1.3333 1.2500

Each element in the first vector is multiplied by the corresponding element in the second vector. As with addition and subtraction, the vectors must be the same shape. Note what happens if we make one of the two vectors tall and thin (using the ' transpose). We get an error stating that we are trying to multiply two things of different shapes.
Try reorienting V1 so they multiply successfully. Now do it by reorienting V2.

Here's another example.

```matlab
M1=[1 2 3; 4 5 6];
V1.*M1
```

This multiplication is never going to happen for you, regardless of how you transpose M1 and V1. These two variables will never be the same size.

### 3.5 Inner product (dot product)

The second kind of multiplication, the *inner-product* or *dot product* is a bit more complicated. Here, when multiplying matrices $C=A*B$ the **number of columns in A need to match the number of rows in B**.

```matlab
clear all
V1= [1 2 4];
V2= [1.1 2.2 3.3];
whos
```

Both V1 and V2 have one row and three columns. There are various inner products that can be calculated from these matrices or their transposed versions. The first is calculated by transposing the vectors so V1 has a single column, and V2 has a single row. Remember, the number of columns in A must match the number of rows in B.

```
V1'*V2
```
The second is calculated by transposing the vectors so A has three columns and B has three rows.

\[ \mathbf{v1}' \cdot \mathbf{v2}' \]

\[ \text{ans} = 18.7000 \]

The third and fourth are calculated by reversing the order of A and B

\[ \mathbf{v2}' \cdot \mathbf{v1} \]

\[ \mathbf{v2} \cdot \mathbf{v1}' \]

Note that in both cases the answer you get from the calculation \( \mathbf{B}' \cdot \mathbf{A} \) is the transpose of the answer you get from the answer \( \mathbf{A}' \cdot \mathbf{B} \).

You can't do the following calculations, because in these cases the number of rows in B doesn't match the number of columns in A:

\[ \mathbf{v1} \cdot \mathbf{v2} \]

\[ \mathbf{v2} \cdot \mathbf{v1} \]

Now try doing the analogous operations using division.

### 3.6 Matrix multiplication and division

An analogous type of multiplication and division occurs when A and B are matrices. It is still possible to calculate the inner product \( \mathbf{C} = \mathbf{A} \cdot \mathbf{B} \) but the number of columns in A still needs to match the number of rows in B. The size of the output depends on how you multiply the output. If A is m-by-p and B is p-by-n, their product C is m-by-n. I.e. C has the same number of rows as A and the same number of columns as B.

\[ \mathbf{M1} = \mathbf{v1}' \cdot \mathbf{v2} \]
\[
M2 = \begin{bmatrix} 1 & 2 & 3; 4 & 5 & 6; 7 & 8 & 9; 1 & 2 & 3 \end{bmatrix}
\]

\[
M2 = \\
\begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
1 & 2 & 3 \\
\end{array}
\]

\textbf{whos}

\begin{tabular}{lllll}
\textbf{Name} & \textbf{Size} & \textbf{Bytes} & \textbf{Class} & \textbf{Attributes} \\
M1 & 3x3 & 72 & double & \\
M2 & 4x3 & 96 & double & \\
V1 & 1x3 & 24 & double & \\
V2 & 1x3 & 24 & double & \\
\textbf{ans} & 1x1 & 8 & double & \\
\end{tabular}

M1 has 3 rows and 3 columns. M2 has 4 rows and 3 columns. That means we can calculate the following inner products.

\[
M1 * M2'
\]

\[
\text{ans} = \\
\begin{bmatrix} 15.4000 & 35.2000 & 55.0000 & 15.4000 \\
& 30.8000 & 70.4000 & 110.0000 & 30.8000 \\
& 61.6000 & 140.8000 & 220.0000 & 61.6000 \\
\end{bmatrix}
\]

\[
M1' * M2'
\]

\[
\text{ans} = \\
\begin{bmatrix} 18.7000 & 41.8000 & 64.9000 & 18.7000 \\
& 37.4000 & 83.6000 & 129.8000 & 37.4000 \\
& 56.1000 & 125.4000 & 194.7000 & 56.1000 \\
\end{bmatrix}
\]

\[
M2 * M1
\]

\[
\text{ans} = \\
\begin{bmatrix} 18.7000 & 37.4000 & 56.1000 \\
& 41.8000 & 83.6000 & 125.4000 \\
& 64.9000 & 129.8000 & 194.7000 \\
& 18.7000 & 37.4000 & 56.1000 \\
\end{bmatrix}
\]

\[
M2 * M1'
\]

\[
\text{ans} = \\
\begin{bmatrix} 18.7000 & 37.4000 & 56.1000 \\
& 41.8000 & 83.6000 & 125.4000 \\
& 64.9000 & 129.8000 & 194.7000 \\
& 18.7000 & 37.4000 & 56.1000 \\
\end{bmatrix}
\]
The following multiplications aren't allowed because the number of columns in M1 doesn't match the number of rows in M2.

\[ M1 \times M2 \]

\[ M2' \times M1 \]

??? Error using ==> mtimes
Inner matrix dimensions must agree.

3.7 More Calculation Stuff

To raise 10 to the 5th power \((10^5)\) you simply do the following:

\[ y=10^5 \]

To do this with vectors you need to add the period. You need to use the period regardless of whether you are taking a vector and raising it to a single number, taking a single number and raising it to a vector, or taking a vector and raising it to another vector (which will need to be of the same size and shape).

\[ y=3; \ y^2 \]

\[ \text{ans} = 9 \]

\[ [1:5].^2 \]
\[ y=2.^[1:5] \]

\[ y = \begin{array}{cccccc} 2 & 4 & 8 & 16 & 32 \end{array} \]

\[ y=[2:6].^[1:5] \]

You can take either the natural or the base 10 log of a number. The default is the natural log.

\[ y=\log(10) \]
\[ y=\log10(10) \]
\[ y=\log([5 \ 6 \ 7 \ 8]) \]

The exponential function \( e^x \) is simply:
\texttt{y=exp(1:3)}
\begin{verbatim}
y =  
   2.7183   7.3891  20.0855
\end{verbatim}

Other useful commands are \texttt{round}, \texttt{min} and \texttt{max}.

\texttt{round(3.14)}
\begin{verbatim}
ans =  
   3
\end{verbatim}

\texttt{x=1:5; min(x)}
\begin{verbatim}
ans =  
   1
\end{verbatim}

\texttt{max(x)}
\begin{verbatim}
ans =  
   5
\end{verbatim}

When \texttt{min} and \texttt{max} are used on a matrix they give you the minimum along each column, not the minimum of the entire matrix. The command \texttt{max} works in a similar way.

\texttt{mat=[1 2 3 4 5; 1.1 2.2 3.3 4.4 5.5];}
\texttt{min(mat)}
\begin{verbatim}
ans =  
   1    2    3    4    5
\end{verbatim}

This is a good time to begin to start using \texttt{doc} (the Matlab help). Try looking up these commands using \texttt{doc} and reading them, e.g.

\texttt{>>doc round}

Once the doc window is open you can just search for them by typing them directly into the search line there.

\textbf{3.8 Multidimensional matrices}

Matrices can be 3, 4 or more dimensions, though some commands won’t work for matrices with more than 2 or 3 dimensions, as you will discover later in this book.
There are many circumstances where a three-dimensional matrix is useful in the behavioral sciences. One of the most common examples is when you want to present subjects with a series of images over time. In that case it is very natural to use the first and second dimensions to represent each image, and the third dimension to represent time. In that case, if you wanted to look at the first image presented in time you would reference it as follows:

mat(:, :, 1)

ans =
    0     1     1     0
    0     0     0     0
    0     0     0     1
    0     1     0     0
    0     0     1     0

The third image in time would be:

mat(:, :, 3)

ans =
    0     0     0     0
    1     1     0     1
    1     0     0     1
    0     0     0     0
    0     0     0     0

Figure 3.4 A three dimensional matrix.
If instead, you wanted to look at what was happening in the top left corner of the screen over time, you would reference it as follows:

```matlab
mat(1, 1, :)
```

```matlab
ans(:,:,1) =
0
ans(:,:,2) =
1
ans(:,:,3) =
0
```

If you wanted to know the values in the bottom line of the screen in the first image you would reference it as follows:

```matlab
mat(3, :, 1)
```

```matlab
ans =
0 0 0 1
```

Finally, if you wanted to know what was happening along the bottom line of the screen over time.

```matlab
mat(3, :, :)
```

```matlab
ans(:,:,1) =
0 0 0 1
ans(:,:,2) =
0 0 0 0
ans(:,:,3) =
0 0 0 0
```

Here's an example of how a four dimensional matrix might be used to represent data. Functional magnetic resonance data is generally represented in four dimensions. The first two dimensions each represent a single image (slice) through the brain. The third dimension represents all the slices needed to represent the entire brain. The fourth dimension represents the time course of the experiment. You can think of it as multiple cubes over time (or not). Here's a fake fmri dataset with 64x64 voxels in the slice, 28 slices and 80 time points.

```matlab
fakefmri=randn(64,64, 28, 80);
ontimepts=[1:10, 21:30, 31:40, 41:50, 61:70];
```
fakefmri(5:10, 5:10, 5:8, ontimepts) = fakefmri(5:10, 5:10, 5:8, ontimepts) + 1;

First let's look at a single voxel over time (we'll talk more about plot later). squeeze is a command that removes any singleton dimensions.

size(fakefmri(6, 6, 7, :))

ans =
   1     1     1    80

size(squeeze(fakefmri(6, 6, 7, :)))

ans =
   80     1

plot(1:80, squeeze(fakefmri(6, 6, 7, :)))

Or we can look at a single slice at a single time point (again, you'll learn more about imagesc later).

imagesc(fakefmri(:, :, 7, 22))
Or look at a single slice averaged across the ontimepts. Note that mnfakefmri is a 3d vector, since we have averaged over time.

\[
\text{mnfakefmri} = \text{mean}(\text{fakefmri}(:, :, :, \text{ontimepts}), 4);
\]
\[
\text{imagesc} (\text{mnfakefmri}(:, :, 7))
\]

3.9 Making matrices.m

% MakingMatrices.m
% This program provides examples of cunning ways to create matrices
% Written IF 3/2007
mat1=zeros(5, 4);

The command \texttt{zeros(5, 4)} creates a matrix containing all zeros with five rows and four columns. Remember that the first dimension in a matrix is the rows, and the second dimension is the columns. Also try using the command \texttt{ones} (which works in a similar way) to make a matrix with 7 rows and 3 columns. Other commands that work in a similar way are \texttt{rand} (which fills the matrix with random values varying between 0-1) and \texttt{randn} (which fills the matrix with normally distributed noise with zero mean and unity variance). In this next example you are setting all the rows in the third column to be 1.

\[
\text{mat2}=\text{mat1};\text{mat2}(1:5, 3)=1
\]

\[
\text{mat2} = \\
0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 \\
\]

Another way of writing this command would be as follows, here instead of having to know that \texttt{mat2} has five rows, we use the expression 1:end to tell Matlab to use the vector that goes from the first row to the last row.

\[
\text{mat2}=\text{mat1};
\]
mat2(1:end,3)=6

mat2 =
0 0 6 0
0 0 6 0
0 0 6 0
0 0 6 0
0 0 6 0

Finally, here's an even more succinct shortcut where we simply use a colon to tell Matlab to include all rows.

    mat2=mat1;
    mat2(:, 3)=7

mat2 =
0 0 7 0
0 0 7 0
0 0 7 0
0 0 7 0
0 0 7 0

Now let's set all the columns in the fourth row to be 1. The colon now means to include the entire column.

    mat3=mat1;
    mat3(4, 1:4)=1

mat3 =
0 0 0 0
0 0 0 0
0 0 0 0
1 1 1 1
0 0 0 0

Other ways of writing this command would be:

    mat3(4, 1:end)=1;
    mat3(4, :)=1

mat3 =
0 0 0 0
0 0 0 0
0 0 0 0
1 1 1 1
0 0 0 0

Here we create a matrix that is four by four zeros (it is a weirdness of the zeros command that the convention is simply giving a single argument (4) makes a 4x4 square matrix).
```matlab
dmat4 = zeros(4)
dmat4 =
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
```

Then we go through each row of the matrix, and replace the place in the matrix that is the $i$-th row and $i$-th column with the number $i$. We are doing this using a `for` loop. Matlab will go through the loop four times. Each time it goes through the loop it will wait at the `pause` command for you to press a key. The first time it goes through the loop $i$ will be equal to 1, so the command `mat4(i, i)=i;` will be the equivalent of `mat4(1, 1)=1`. The second time it goes through the loop $i$ will be 2, so `mat4(2, 2)=2`, and so on up to $i=4$, `mat4(4, 4)=4`. I've left the semicolon off so you can watch `mat4` gradually change.

```matlab
dmat4 = zeros(4);
dfor i=1:4
d  dmat4(i, i)=i
  dpause
dend
dmat4 =
1 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
dmat4 =
1 0 0 0
0 2 0 0
0 0 0 0
0 0 0 0
dmat4 =
1 0 0 0
0 2 0 0
0 0 3 0
0 0 0 0
dmat4 =
1 0 0 0
0 2 0 0
0 0 3 0
0 0 0 4
```

In this example we sequentially go through five rows of a matrix filling the columns with a vector that depends on which row it is.

```matlab
dmat5 = zeros(6)
dfor i=1:6
dmat5(i,:)=[-2 0 -1 1 2 3]
d  dpause
dend```
As we go through the loop six times, we filled the columns one through six of mat5 with a vector that was the values -2:3. Next, try doing the same thing but using the vector [-1 -2 0 -1 1 2 3].

```matlab
mat5=zeros(6);
for i=1:6
    mat5(i,:)=[-1 -2 0 -1 1 2 3]
```
Subscripted assignment dimension mismatch?

This gives an error because you are trying to squeeze a vector that is 1 row and 5 columns into the row of a matrix that only has four columns. You often get this error because one of your vectors or matrices is the wrong size and/or shape, as in the examples here. The subscripted assignment dimension error means you are trying to fit a square peg into a round hole.

Here, as we go through the loop six times, we fill columns one through six of mat5 with the vector \(-2:3\).
Now try this using the vector \([-1 -2 0 -1 1 2 3]\).

```matlab
mat5=zeros(6);
for i=1:6
    mat5(i,:)=[-1 -2 0 -1 1 2 3]
pause
end
```

?? Subscripted assignment dimension mismatch.

You get an error! This is because you are trying to squeeze a vector that is 1 row and seven columns \([-1 -2 0 -1 1 2 3]\) into the row of a matrix that only has six columns.

**Figure 3.6** Zachary Boynton Fine demonstrating the subscripted assignment dimension error.

Here are some other examples of dimension mismatch errors.

```matlab
x=zeros(4)
x(2, :) = 1:5
```

```matlab
x =
    0     0     0     0
    0     0     0     0
    0     0     0     0
    0     0     0     0
```

??? Subscripted assignment dimension mismatch.

```matlab
x=zeros(2, 4)
y=ones(4, 2)
x(1, :)= y(1, :)
```

```matlab
x =
    0     0     0     0
    0     0     0     0
y =
    1     1
    1     1
    1     1
    1     1
```

??? Subscripted assignment dimension mismatch.

```matlab
x=zeros(2, 4)
y=ones(5, 5)
x(:, :)=y
```

```matlab
x =
    0     0     0     0
    0     0     0     0
y =
```
Weirdly, the following command, which seems identical, will work. Check the Wha? box to understand why.

```matlab
x=zeros(2, 4);
y=ones(5, 5);
x(1:5, 1:5)=y
```

Here's yet another matrix. Here we are saying we want to place ones in the matrix in the positions where the rows are between two through five and the columns are between one and three.

```matlab
mat6=zeros(6);
mat6(2:5, 1:3)=1
```

In the next example, what we put into the matrix on each iteration of the loop depends on the value of i. When i is one, then the vector [1 1 2 2 1] is put into the first row of mat7. When i is two, then the vector [2 2 3 3 2] is put into the second row of mat7 and so on.

```matlab
mat7=zeros(5);
for i=1:5
    mat7(i, :)=[0 0 1 1 0]+i;
end
mat7
```

mat7 =

```
1 1 2 2 1
```
Now let's do the same thing but we'll put the vectors in along the columns instead of the rows. All we need to do is change `mat7(i, :)` to `mat7(:, i)`.

```matlab
mat7=zeros(5);
for i=1:5
    mat7(:, i)= [0 0 1 1 0]+i;
end
mat7
```
```
1     2     3     4     5
1     2     3     4     5
2     3     4     5     6
2     3     4     5     6
1     2     3     4     5
```

Technically we're trying to vector that has one row and five columns into a space that has five rows and one columns. So if Matlab wanted to be pedantic it would give the 'Subscripted assignment dimension mismatch error'. But Matlab, rather tolerantly, will let this one go. It would be better style to make the two vectors match in shape by transposing the right hand side as follows:

```matlab
mat7(:, i)=[0 0 1 1 0]'+i
```

In the next example we are using a nested loop. Basically it is one for loop inside another. So in this example we use `i` go down each row. For each value of `i` we then use `j` to go through each column in turn.

```matlab
mat8=zeros(3, 4);
for i=1:3
    for j=1:4
        mat8(i, j)=i+j
    end
end
mat8
```
```
2     0     0     0
0     0     0     0
0     0     0     0
```
```
2     3     0     0
0     0     0     0
0     0     0     0
```
```
2     3     4     0
0     0     0     0
0     0     0     0
```
Here's a tricky one. The expression \((i-1)*4\)+\(j\) means that when we are on the first row, the columns are labeled as 1,2,3,4. When we are on the second row the columns are labeled as 5,6,7,8 and so on … When \(i=1\) then \((i-1)*4\)=0, so the columns are filled with the value of \(j\). When \(i=2\) then \((i-1)*4\)=4, so the columns are filled with the value of \(j+4\), and so on.

```matlab
mat9=zeros(3,4);
for i=1:3
    for j=1:4
        mat9(i, j)=((i-1)*4)+j
    end
end

mat9 =
1  0  0  0
0  0  0  0
0  0  0  0
```
In fact, you don't do a nested loop to do this – here's how to do this with a single loop.

```matlab
mat9=zeros(3,4);
for i=1:3
    mat9(i, :) = ((i-1)*4)+[1:4]
end
```

mat9 =

1 2 3 4
0 0 0 0
0 0 0 0
Being able to do these sorts of manipulations is one of the keys to being able to write good code in Matlab, so take your time and make sure you understand how these examples work. Also, make sure you can do all the exercises before you move onto the next chapter.

### 3.10 sub2ind and ind2sub

Sometimes it’s useful to convert back and forth between vectors and matrices.

```matlab
mat=[1 2 3; 4 5 6; 7 8 9];
vect=mat(:);
mat=[1 2 3; 4 5 6; 7 8 9]
vect=mat(:);

mat =
    1     2     3
    4     5     6
    7     8     9

The colon (the colon has many purposes in Matlab) tells Matlab to unwrap the matrix mat out to be a vector. Note that the unwrapped vector first lists all the rows in the first column, then lists all the rows in the second column, and so on … Suppose you wanted to know where the number eight would be in the vector? Well, the number eight appears in the third row and the second column of the matrix. The 3rd row and 2nd column are known as the row and column *subscripts* of the matrix mat. The `sub2ind` command below calculates the *index* in vect corresponding to the 3rd row and 2nd column (the subscripts) of mat. You simply need to tell it the size of mat, and the row and columns subscripts.

```matlab
ind=sub2ind(size(mat), 3, 2);
```

```matlab
ind =
    6
```

So the 3rd row and 2nd column in the matrix corresponds to the 6th *index* into the vector. So both of the following should give you the number eight.

```matlab
vect(ind)
```

```matlab
ans =
    8
```

```matlab
mat(3, 2)
```

```matlab
ans =
    8
```
You can also go the other way using `ind2sub`. Once again you need to provide the size of the matrix and the position in the vector that you want to find the matrix subscripts for.

```matlab
[sub_row, sub_col] = ind2sub(size(mat), 6);
sub_row
```

```
sub_row = 3
```

```matlab
sub_col
```

```
sub_col = 2
```

### 3.11 Logical operations

These are ways of checking the truth of statements. In programming truth is expressed as being 1 and falsity is 0. Note that you use a single `=` to assign a value to a variable. You use a double equal `==` to determine whether the value of the left hand side variable is equal to that of the right hand side variable. This expression gives you a 0 because the statement is false.

```matlab
n1=2; n2=4;
n1==3.2
```

```
ans = 0
```

```matlab
n1==n2
```

```
ans = 0
```

This expression is true.

```matlab
n1=4; n1==n2
```

```
ans = 1
```

You can also see whether one number is *not* equal to another.

```matlab
n1=4; n2=4; n1~=n2
```

```
ans = 0
```
This is false.

\[ n_2 = 5; \ n_1 \sim n_2 \]
\[ \text{ans} = 1 \]

This is true. One common way of using logical truth operations is in an if statement. An if statement checks whether the statement following the if is true or not. If the statement is true, Matlab carries out the operations between the if and the end. If the statement is false Matlab doesn't carry out those operations.

\[ n_1 = 3.2 \]
\[ \text{if round}(n_1) = n_1 \]
\[ \text{disp}('n is a round number'); \]
\[ \text{end} \]
\[ n_1 = 3.2000 \]

You can also tell Matlab what to do if the statement is false, using an else.

\[ n = 3.2 \]
\[ \text{if round}(n_1) = n_1 \]
\[ \quad \text{disp}('n is a round number'); \]
\[ \text{else} \]
\[ \quad \text{disp}('n is not a round number'); \]
\[ \text{end} \]
\[ n = 3.2000 \]
\[ n \text{ is not a round number} \]

Traditionally the statement that is evaluated by an if statement is either a 1 or a 0 (true or false) but in Matlab commands following an if statement will always be executed unless the condition following the if results in a 0.

\[ n = -1 \]
\[ \text{if } n \]
\[ \quad \text{disp}('hi there cutie-pants') \]
\[ \text{else} \]
\[ \quad \text{disp}('bye-bye darling') \]
\[ \text{end} \]
\[ n = -1 \]
\[ \text{hi there cutie-pants} \]
The `and` `&&` and `or` `||` operators are used when you want to carry out a loop only when more than one condition is true (`&`) or when either of two conditions is true (`|`). Try the following:

```matlab
clear all;
n1=1; n2=2; n3=3;
n1<n2 && n1>n3
ans = 0

n1<n2 || n1>n3
ans = 1
```